SOMMAIRE

Algorithmie

INTRODUCTION . . . . . *P 2-3*

CHAPITRE 1 : Les variables . . . . . *P 4-8*

CHAPITRE 2 : Lecture / Ecriture . . . . . *P 9-10*

CHAPITRE 3 : Les tests *P11-15*

CHAPITRE 4 : Les boucles *P16-18*

CHAPITRE 5 : Les tableaux *P19-21*

CHAPITRE 6 : Les tableaux multidimensionnels *P22-23*

CHAPITRE 7 : Les fonctions prédéfinies *P24-25*

CHAPITRE 8 : Les fichiers séquentiels. *P26-29*

CHAPITRE 9 : Procédures et fonctions *P30-34*

CHAPITRE 10 : Notions complémentaires *P35-36*

INTRODUCTION

Préambule

L'algorithmique est un terme d'origine arabe, comme algèbre, amiral ou zénith.

Ainsi, l'algo n'est pas " rythmique ", à la différence du bon rock'n roll. L'algo n'est pas non plus " l'agglo ".

Alors, ne confondez pas l'algorithmique avec l'agglo rythmique, qui consiste à poser des parpaings en cadence.

1. Qu'est-ce que l'algomachin ?

Avez-vous déjà ouvert un livre de recettes de cuisine ? Avez vous déjà déchiffré un mode d'emploi traduit directement du coréen pour faire fonctionner un magnétoscope ou un répondeur téléphonique réticent ? Si oui, sans le savoir, vous avez déjà exécuté des algorithmes.

Plus fort : avez-vous déjà indiqué un chemin à un touriste égaré ? Avez vous fait chercher un objet à quelqu'un par téléphone ? Ecrit une lettre anonyme stipulant comment procéder à une remise de rançon ? Si oui, vous avez déjà fabriqué – et fait exécuter – des algorithmes.

Comme quoi, l'algorithmique n'est pas un savoir ésotérique réservé à quelques rares initiés touchés par la grâce divine, mais une aptitude partagée par la totalité de l'humanité. Donc, pas d'excuses…

2. Définissons l'algorithmique

Un algorithme, c'est une suite d'instructions, qui une fois exécutée correctement, conduit à un résultat donné. Si l'algorithme est juste, le résultat est le résultat voulu, et le touriste se retrouve là où il voulait aller. Si l'algorithme est faux, le résultat est, disons, aléatoire, et décidément, cette saloperie de répondeur ne marche pas.

Complétons toutefois cette définition. Après tout, en effet, si l'algorithme, comme on vient de le dire, n'est qu'une suite d'instructions menant celui qui l'exécute à résoudre un problème, pourquoi ne pas donner comme instruction unique : " résous le problème ", et laisser l'interlocuteur se débrouiller avec ça ? A ce tarif, n'importe qui serait champion d'algorithmique sans faire aucun effort. Pas de ça Lisette, ce serait trop facile.

Le malheur (ou le bonheur, tout dépend du point de vue) est que justement, si le touriste vous demande son chemin, c'est qu'il ne le connaît pas. Donc, si on n'est pas un goujat intégral, il ne sert à rien de lui dire de le trouver tout seul. De même les modes d'emploi contiennent généralement (mais pas toujours) un peu plus d'informations que " débrouillez vous pour que ça marche ".

Pour fonctionner, **un algorithme doit donc contenir uniquement des instructions compréhensibles par celui qui devra l'exécuter**. C'est d'ailleurs l'un des points délicats pour les rédacteurs de modes d'emploi : les références culturelles, ou lexicales, des utilisateurs, étant variées, un même mode d'emploi peut être très clair pour certains et parfaitement abscons pour d'autres.

En informatique, heureusement, il n'y a pas ce problème : les choses auxquelles ont doit donner des instructions sont les ordinateurs, et ceux-ci ont le bon goût d'être tous strictement aussi idiots les uns que les autres.

3. Faut-il être matheux pour être bon en algorithmique ?

Je consacre quelques lignes à cette question, car cette opinion aussi fortement affirmée que faiblement fondée sert régulièrement d'excuse : " moi, de toute façon, je suis mauvais(e) en algo, j'ai jamais rien pigé aux maths ". Faut-il être " bon en maths " pour expliquer correctement son chemin à quelqu'un ? Je vous laisse juge.

La maîtrise de l'algorithmique requiert deux qualités :

- il faut être rigoureux. Car chaque fois qu'on écrit une série d'instructions qu'on croit justes, il faut se mettre à la place de la machine qui va les exécuter, pour vérifier si le résultat obtenu est bien celui que

l'on voulait. Mais cette opération ne requiert pas la moindre once d'intelligence, uniquement d'être méthodique.

- il faut avoir une certaine intuition, car aucune recette ne permet de savoir a priori quelles instructions permettront d'obtenir le résultat voulu. C'est là, si l'on y tient, qu'intervient la forme " d'intelligence " requise pour l'algorithmique. Alors, c'est certain, il y a des gens qui ont au départ davantage cette intuition que les autres. Mais… mais d'une part, les réflexes, cela s'acquiert, et l'expérience finit par compenser largement bien des intuitions. D'autre part, pour être bon en algo, il ne faut pas oublier le point numéro 1.

4. L'ADN, les Shadoks, et les ordinateurs

Quel rapport me direz-vous ? Eh bien le point commun est : quatre mots de vocabulaire.

L'univers lexical Shadok, c'est bien connu, se limite aux termes " Ga ", " Bu ", " Zo ", et " Meu ". Ce qui leur a tout de même permis de formuler quelques fortes maximes, telles que : " *Mieux vaut pomper et qu'il ne se passe rien, plutôt qu'arrêter de pomper et risquer qu'il se passe quelque chose de pire* " .

L'ADN, qui est en quelque sorte le programme génétique, l'algorithme à la base de construction des êtres vivants, est une chaîne construite à partir de quatre éléments invariables. Ce n'est que le nombre de ces éléments, et l'ordre dans lequel ils sont arrangés, qui va déterminer si on obtient une puce ou un éléphant. Et tous autant que nous sommes, splendides réussites de la Nature, avons été construits par un " programme " constitué uniquement de ces quatre briques, ce qui devrait nous inciter à la modestie.

Enfin, les ordinateurs, quels qu'ils soient, ne sont fondamentalement capables d'exécuter que quatre opérations logiques (je n'inclus pas ici les opérations de calcul). Ces opérations sont :

- l'affectation de variables
- la lecture / écriture
- les tests
- les boucles

Un algorithme informatique se ramène donc toujours au bout du compte à la combinaison de ces quatre petites briques de base. Il peut y en avoir quelques unes, quelques dizaines, et jusqu'à plusieurs centaines de milliers dans certains programmes de gestion. Rassurez-vous, dans le cadre de ce cours, nous n'irons pas jusque là (cependant, la taille d'un algorithme ne conditionne pas en soi sa complexité : de longs algorithmes peuvent être finalement assez simples, et de petits très compliqués).

5. Algorithmique et programmation

Pourquoi apprendre l'algorithmique pour apprendre à programmer ? En quoi a-t-on besoin d'un langage spécial, distinct des langages de programmation compréhensibles par les ordinateurs ?

Parce que l'algorithmique exprime les instructions résolvant un problème donné **indépendamment des particularités de tel ou tel langage**. Pour prendre une image, si un programme était une dissertation, l'algorithmique serait le plan, une fois mis de côté la rédaction et l'orthographe. Or, vous savez qu'il vaut mieux faire d'abord le plan et rédiger ensuite que l'inverse…

6. Avec quelles conventions écrit-on un algorithme ?

Historiquement, plusieurs types de notations ont représenté des algorithmes.

Il y a eu notamment une représentation graphique, avec des carrés, des losanges, etc. qu'on appelait des **organigrammes**. Aujourd'hui, cette représentation est quasiment abandonnée, pour deux raisons. D'abord, parce que dès que l'algorithme commence à grossir un peu, ce n'est plus pratique du tout du tout. Ensuite parce que cette représentation favorise le glissement vers un certain type de programmation, dite non structurée (nous définirons ce terme plus tard), que l'on tente en général d'éviter.

C'est pourquoi on utilise généralement une série de conventions appelée " pseudo-code ", qui ressemble à un langage authentique, la plupart des problèmes de syntaxe étant mis de côté.

I Les variables

1. Déclaration

Dans un programme informatique, on va avoir en permanence besoin de stocker provisoirement des valeurs. Il peut s'agir de données issues du disque dur, fournies par l'utilisateur (frappées au clavier), ou que sais-je encore. Il peut aussi s'agir de résultats obtenus par le programme, intermédiaires ou définitifs. Ces données peuvent être de plusieurs types (on en reparlera) : elles peuvent être des nombres, du texte, etc. Toujours est-il que dès que l'on a besoin de stocker une information dans un programme, on utilise une **variable**.

Pour employer une image, une variable est une boîte, repérée par une étiquette. Pour avoir accès au contenu de la boîte, il suffit de la désigner par son étiquette.

En réalité, dans la mémoire vive de l'ordinateur, il n'y a ni boîte ni étiquette collée dessus. Il y a un emplacement de mémoire, désigné par une adresse binaire. Le langage informatique se charge, entre autres rôles, de vous épargner la gestion fastidieuse de ces emplacements mémoire et de leurs adresses. Il est beaucoup plus facile d'employer les étiquettes de son choix, que de devoir manier des adresses binaires.

La première chose à faire avant de pouvoir utiliser une variable est de créer la boîte et de lui coller une étiquette. Ceci se fait tout au début de l'algorithme, avant même les instructions proprement dites. C'est ce qu'on appelle la **déclaration des variables**. C'est un genre de déclaration certes moins romantique qu'une déclaration d'amour, mais d'un autre côté moins désagréable qu'une déclaration d'impôts.

Le **nom** de la variable (l'étiquette de la boîte) obéit à des impératifs changeant selon les langages. Toutefois, une règle absolue est qu'ils peuvent comporter des lettres et des chiffres, mais qu'ils excluent la plupart des signes de ponctuation, en particulier les espaces. Un nom de variable correct commence impérativement par une lettre. Quant au nombre maximal de signes pour un nom de variable, il dépend du langage utilisé.

En pseudo-code algorithmique, on est bien sûr libre du nombre de signes pour un nom de variable, même si pour des raisons purement pratiques, et au grand désespoir de Stéphane Bern, on évite généralement les noms à rallonge.

2. Type des variables

Il ne suffit pas de créer une boîte (réserver un emplacement mémoire) ; encore doit-on préciser ce que l'on voudra mettre dedans, car de cela dépendent la taille de la boîte (de l'emplacement mémoire) et le **type** de codage utilisé.

2.1 Types numériques classiques

Commençons par le cas le plus fréquent, celui d'une variable destinée à recevoir des nombres.

Si l'on réserve un octet pour coder un nombre, on ne pourra coder que 2^8 = 256 valeurs différentes. Cela peut signifier par exemple les nombres entiers de 1 à 256, ou de 0 à 255, ou de –127 à +128, c'est une pure question de convention. Mais ce qui n'est pas une convention, c'est qu'on est limité à 256 valeurs possibles. Si l'on réserve deux octets, on a droit à 65 536 valeurs ; avec trois octets, 16 777 216, etc. Et là se pose un autre problème : ce codage doit-il représenter des nombres décimaux ? des nombres négatifs ?

Bref, le type de codage (autrement dit, le type de variable) choisi pour un nombre va déterminer :

- les valeurs maximales et minimales des nombres pouvant être stockés dans la variable
- la précision de ces nombres (dans le cas de nombres décimaux).

Tous les langages, quels qu'ils soient offrent un " bouquet " de types numériques, dont le détail est susceptible de varier légèrement d'un langage à l'autre. Grosso modo, on retrouve cependant les types suivants :

Type Numérique	Plage

Byte (octet)	0 à 255
Entier simple	-32 768 à 32 767
Entier long	-2 147 483 648 à 2 147 483 647
Réel simple	-3,40E38 à -1,40E-45 pour les valeurs négatives 1,40E-45 à 3,40E38 pour les valeurs positives
Réel double	1,79E308 à -4,94E-324 pour les valeurs négatives 4,94E-324 à 1,79E308 pour les valeurs positives

Pourquoi ne pas déclarer toutes les variables numériques en réel double, histoire de bétonner et d'être certain qu'il n'y aura pas de problème ? En vertu du principe de l'économie de moyens. Un bon algorithme ne se contente pas de marcher ; il marche en évitant de gaspiller les ressources de la machine. Sur certains programmes de grande taille, l'abus de variables surdimensionnées peut entraîner des ralentissements notables à l'exécution, voire un plantage pur et simple. Alors, autant prendre de bonnes habitudes d'hygiène…

Une déclaration algorithmique de variables aura ainsi cette tête :

Variable g **en Entier Long**

Variables PrixHT, TauxTVA, PrixTTC **en Réel Simple**

2.2 Autres types numériques

Certains langages autorisent d'autres types numériques, notamment :

- le type **monétaire** (avec strictement deux chiffres après la virgule)
- le type **date** (jour / mois / année).

Nous n'emploierons pas ces types ici, mais vous ne manquerez pas de les rencontrer en programmation proprement dite.

2.3 Types non numériques

Fort heureusement, les boîtes que sont les variables peuvent contenir bien d'autres informations que des nombres. Sans cela, on serait un peu embêté dès que l'on devrait stocker un nom de famille, par exemple.

On dispose donc également du **type alphanumérique** (également appelé **type caractère**) : dans une variable de ce type, on stocke des caractères, qu'il s'agisse de lettres, de signes de ponctuation, d'espaces, ou de chiffres. Le nombre maximal de caractères pouvant être stockés dans une seule variable dépend du langage utilisé. Une série de caractères (stockée ou non dans une variable alphanumérique, d'ailleurs), s'appelle une **chaîne** de caractères. Et une telle chaîne de caractères est toujours notée entre guillemets. Pourquoi diable ? Parce que 423 peut représenter le nombre 423, ou la suite de caractères 4, 2, et 3, selon le type de codage (le type de la variable) qui a été utilisé pour le stocker. Les guillemets permettent d'éviter toute ambiguïté à ce sujet. On y reviendra.

Un autre type est le type **booléen** : on y stocke uniquement les valeurs logiques VRAI et FAUX.

3. L'instruction d'affectation

3.1 Syntaxe et signification

Ouf, après tout ce baratin préliminaire, on aborde enfin nos premières véritables manipulations d'algo. Pas trop tôt.

La seule chose qu'on puisse vraiment faire avec une variable, c'est de **l'affecter, c'est-à-dire lui attribuer une valeur**. En algorithmique, cette instruction se note avec le signe ⇦.

Ainsi :

Toto ⇦24

Attribue la valeur 24 à la variable Toto. Ce qui, soit dit en passant, sous-entend impérativement que Toto est une variable de type numérique.

On peut attribuer à une variable la valeur d'une autre variable, telle quelle ou modifiée. Par exemple :

Tutu ⇦ Toto

Signifie que la valeur de Tutu est maintenant celle de Toto.

Notez que cette instruction n'a modifié en rien la valeur de Toto : une instruction d'affectation ne modifie que ce qui est situé à gauche de la flèche.

Tutu ⇦ Toto + 4

Si Toto contenait 12, Tutu vaut maintenant 16. De même que précédemment, Toto vaut toujours 12.

Tutu ⇦ Tutu + 1

Si Tutu valait 6, il vaut maintenant 7. La valeur de Tutu est modifiée, puisque Tutu est la variable située à gauche de la flèche.

3.2 Ordre des instructions

Il va de soi que l'ordre dans lequel les instructions sont écrites va jouer un rôle essentiel dans le résultat final. Considérons les deux algorithmes suivant (notez que ce sont les premiers algorithmes complets que nous examinerons, comme quoi on avance) :

Variable A **en Entier Variable** A **en Entier**

Début Début

A ⇦ 34 A ⇦ 12

A ⇦ 12 A ⇦ 34

Fin Fin

Il est clair que dans le premier cas la valeur finale de A est 12, dans l'autre elle est 34 .

Il est tout aussi clair que ceci ne doit pas nous étonner. Lorsqu'on indique le chemin à quelqu'un, dire " prenez tout droit sur 1km, puis à droite " n'envoie pas les gens au même endroit que si l'on dit " prenez à droite puis tout droit pendant 1 km ".

Enfin, il est également clair que si l'on met de côté leur vertu pédagogique, les deux algorithmes ci-dessus sont parfaitement idiots ; à tout le moins ils contiennent une incohérence. Il n'y a aucun intérêt à affecter une variable pour l'affecter différemment juste après. En l'occurrence, on aurait tout aussi bien atteint le même résultat en écrivant simplement :

Variable A **en Entier Variable** A **en Entier**

Début Début

A ⇦ 12 A ⇦ 34

Fin Fin

Eh bien maintenant, à vous de jouer !

3.3 Expressions et opérateurs

Si on fait le point, on s'aperçoit que dans une instruction d'affectation, on trouve :

- à gauche de la flèche, un nom de variable, et uniquement cela.
- à droite de la flèche, ce qu'on appelle une **expression**. C'est-à-dire un ensemble de valeurs liées par des **opérateurs**, et dont le résultat final est obligatoirement du même type que la variable située à gauche. Si l'un de ces points n'est pas respecté, la machine sera incapable d'exécuter l'affectation, et déclenchera une erreur (est-il besoin de dire que si aucun de ces points n'est respecté, il y aura aussi erreur !)

Revenons maintenant sur ce terme d'opérateurs. Un opérateur est un signe qui peut relier deux valeurs, pour produire un résultat. Les opérateurs possibles dépendent du type des valeurs qui sont en jeu. Allons-y, faisons le tour, c'est un peu fastidieux, mais comme dit le sage, quand c'est fait, c'est plus à faire.

Opérateurs numériques :

Ce sont les quatre opérations arithmétiques tout ce qu'il y a de classique.

+	addition
-	soustraction
*	multiplication
/ division	

Mentionnons également le ^ qui signifie " puissance ". 45 au carré s'écrira donc 45 ^ 2.

Enfin, on a le droit d'utiliser les parenthèses, avec les mêmes règles qu'en mathématiques. La multiplication et la division ont " naturellement " priorité sur l'addition. Les parenthèses ne sont ainsi utiles que pour modifier cette priorité naturelle.

Cela signifie qu'en informatique, 12 * 3 + 5 et (12 * 3) + 5 valent strictement la même chose, à savoir 41. Pourquoi dès lors se fatiguer à mettre des parenthèses inutiles ?

En revanche, 12 * (3 + 5) vaut 12 * 8 soit 96. Rien de difficile là-dedans, que du normal.

Opérateur alphanumérique : &

Cet opérateur permet de **concaténer**, autrement dit d'agglomérer, deux chaînes de caractères.

Exemple

Variables A, B, C **en Caractère**

Début

A ← "Gloubi"

B ← "Boulga"

C ← A & B

Fin

La valeur de C à la fin de l'algorithme est "GloubiBoulga"

Opérateurs logiques :

Il s'agit du ET, du OU et du NON. Nous les laisserons de côté… provisoirement, soyez-en sûrs.

4. Deux remarques pour terminer

J'attire votre attention sur la trompeuse similitude de vocabulaire entre les mathématiques et l'informatique. En mathématiques, une " variable " est généralement une inconnue. Lorsque j'écris que $y = 3x + 2$, les " variables " x et y satisfaisant à l'équation existent en nombre infini (graphiquement, l'ensemble des solutions à cette équation dessine une droite). Lorsque j'écris $ax^2 + bx + c = 0$, la " variable " x désigne les solutions à cette équation, c'est-à-dire 0, une ou deux valeurs à la fois… En informatique, une variable a toujours une valeur et une seule. A la rigueur, dans certains langages, mais pas tous, elle peut ne pas avoir de valeur du tout (tant qu'on ne l'a pas affectée). Dans les autres langages, les variables non encore affectées sont considérées comme valant zéro. Et cette valeur justement, ne " varie " pas à proprement parler (du moins ne varie-t-elle que lorsqu'elle est l'objet d'une instruction d'affectation).

La deuxième remarque concerne le signe de l'affectation. En algorithmique, comme on l'a vu, c'est le ⇦ . Mais en pratique, la quasi totalité des langages emploient le signe égal. Et là, pour les débutants, la confusion est facile avec les maths. En maths, A = B et B = A sont deux propositions strictement équivalentes. En informatique, absolument pas, puisque cela revient à écrire A ⇦ B et B ⇦ A, deux choses bien différentes. Donc, attention ! ! !

II Lecture écriture

1. De quoi parle-t on ?

Trifouiller des variables en mémoire vive par un chouette programme, c'est vrai que c'est très rigolo, et on a tous bien ri. Cela dit, à la fin de la foire, on peut tout de même se demander à quoi ça sert.

En effet. Imaginons que nous ayons fait un programme pour calculer le carré d'un nombre, mettons 12. Si on a fait au plus simple, on a écrit un truc du genre :

Exemple

Variable A en **Numérique**

Début

A ← 12^2

Fin

Ce gentil programme nous donne le carré de 12. Mais si l'on veut le carré d'un autre nombre que 12, il faut réécrire le programme. Bof.

D'autre part, le résultat est très certainement calculé par la machine. Mais elle le garde soigneusement pour elle, et le pauvre utilisateur qui fait exécuter ce programme, lui, ne saura jamais quel est le carré de 12. Re-bof.

C'est pourquoi un certain nombre d'instructions permettent à la machine de dialoguer avec l'utilisateur.

Dans un sens, cela permet à l'utilisateur de rentrer des valeurs au clavier pour qu'elles soient utilisées par le programme. Cette opération est la **lecture**.

Dans l'autre sens, cela permet au programme de communiquer des valeurs à l'utilisateur en les affichant à l'écran. Cette opération est **l'écriture**.

2. Une remarque essentielle

A première vue, on peut avoir l'impression que les informaticiens étaient bourrés comme des vaches lorsqu'ils ont baptisés ces opérations : lorsque l'utilisateur doit écrire au clavier, on appelle ça la lecture, et lorsqu'il doit lire sur l'écran on appelle çà l'écriture. Mais avant d'insulter cette digne corporation, il faut réfléchir un peu plus. Un algorithme, cela programme la machine, pas l'utilisateur. Donc quand on dit à la machine de lire une valeur, cela implique que l'utilisateur va devoir l'écrire. Et quand on demande à la machine d'écrire cette valeur, c'est pour que l'utilisateur puisse la lire. Et toc.

3. Les instructions de lecture et d'écriture

Tout bêtement, pour que l'utilisateur entre la (nouvelle) valeur de Titi, on mettra :

Lire Titi

Dès que le programme tombe sur cette instruction, **l'exécution s'arrête** ; l'ordinateur attend sagement que l'utilisateur ait frappé quelque chose au clavier, et dès que la touche Enter a été frappée, l'exécution reprend.

Pour écrire quelque chose à l'écran, c'est aussi simple que :

Ecrire Toto

On peut également choisir d'écrire des **libellés** à l'écran, afin de prévenir l'utilisateur de ce qu'il doit frapper (c'est même assez fortement recommandé) :

Ecrire "Entrez votre nom : "

Lire NomFamille

Notez qu'il y a une différence majeure entre afficher un libellé et le contenu d'une variable :

Exemple

Variable Bonjour en **Caractère Variable** Bonjour en **Caractère**

Début Début

Bonjour ← "Ave Cesar" Bonjour ← "Ave Cesar"

Ecrire Bonjour Ecrire "Bonjour"

Fin Fin

Le premier algorithme affiche le contenu de la variable Bonjour, autrement dit " Ave Cesar ". Le second affiche le libellé " Bonjour ", qui n'a rien à voir avec la variable du même nom.

III Les tests

Je vous avais dit que l'algorithmique, c'est la combinaison de quatre structures élémentaires. Nous en avons déjà vu deux, voici la troisième. Autrement dit, on a quasiment fini le programme.

Mais non, je rigole.

1. De quoi s'agit-il ?

Reprenons le cas de notre " programmation algorithmique du touriste égaré ". Normalement, notre algorithme ressemblera à quelque chose comme : " Allez tout droit jusqu'au prochain carrefour, puis prenez à droite et ensuite la deuxième à gauche, et vous y êtes ".

Mais en cas de doute légitime de votre part, cela pourrait devenir : " Allez tout droit jusqu'au prochain carrefour et là regardez à droite. Si la rue est autorisée à la circulation, alors prenez la et ensuite c'est la deuxième à gauche. Mais si en revanche elle est en sens interdit, alors continuez jusqu'à la prochaine à droite, prenez celle-là, et ensuite la première à droite ".

Ce deuxième algorithme a ceci de supérieur au premier qu'il prévoit, en fonction d'une situation pouvant se présenter de deux façons différentes, deux façons alternatives d'agir. Cela suppose que l'interlocuteur (le touriste) sache analyser la condition que nous avons fixée à son comportement (" la rue est-elle en sens interdit ? ") pour effectuer la série d'actions correspondante.

Eh bien, croyez le ou non, mais les ordinateurs possèdent cette aptitude, sans laquelle d'ailleurs nous aurions bien du mal à les programmer. Nous allons donc pouvoir parler à notre ordinateur comme à notre touriste, et lui donner des séries d'instructions à effectuer selon que la situation se présente d'une manière ou d'une autre.

2. Structure d'un test

Il n'y a que deux formes possibles pour un test ; la forme de gauche est la forme complète, celle de droite la forme simple.

```
Si booléen Alors Si booléen Alors
Instructions 1 Instructions
Sinon Finsi
Instructions 2
Finsi
```

Un booléen est une expression dont la valeur est VRAI ou FAUX. Cela peut donc être (il n'y a que deux possibilités) :

- une variable de type booléen
- une condition

Nous reviendrons dans quelques instants sur ce qu'est une condition en informatique.

Toujours est-il que la structure d'un test est relativement claire. Arrivé à la première ligne (Si…Alors) la machine examine la valeur du booléen. Si ce booléen a pour valeur VRAI, elle exécute la série d'instructions 1. Cette série d'instructions peut être très brève comme très longue, cela n'a aucune importance. A la fin de cette série d'instructions, au moment où elle arrive au mot " Sinon ", la machine sautera directement à la première instruction située après le " Finsi ". De même, au cas où le booléen avait comme valeur " Faux ", la machine saute directement à la première ligne située après le " Sinon " et exécute l'ensemble des " instructions 2 ".

La forme simplifiée correspond au cas où l'une des deux " branches " du Si est vide. Dès lors, plutôt qu'écrire " sinon ne rien faire du tout ", il est plus simple de ne rien écrire.

Exprimé sous forme de pseudo-code, la programmation de notre touriste de tout à l'heure donnerait donc quelque chose du genre :

Exemple

```
Allez tout droit jusqu'au prochain carrefour
Si la rue à droite est autorisée à la circulation Alors
Tournez à droite
Avancez
Prenez la deuxième à gauche
Sinon
Continuez jusqu'à la prochaine rue à droite
Prenez cette rue
Prenez la première à droite
Fin si
```

3. Qu'est ce qu'une condition ?

Une condition est une comparaison. Cette définition est essentielle !

C'est-à-dire qu'elle est composée de trois éléments :

- une valeur
- un **opérateur de comparaison**
- une autre valeur

Les valeurs peuvent être a priori de n'importe quel type (numériques, caractères…)

Les opérateurs de comparaison sont : = <> < > =< >=

L'ensemble constitue donc si l'on veut une affirmation, qui a un moment donné est VRAIE ou FAUSSE.

A noter que ces opérateurs de comparaison s'emploient tout à fait avec des caractères. Ceux-ci sont codés par la machine dans l'ordre alphabétique, les majuscules étant systématiquement placées avant les minuscules. Ainsi on a :

"t" < "w" VRAI

"Maman" > "Papa" FAUX

"maman" > "Papa" VRAI

Attention à certains raccourcis du langage courant qui peuvent mener à des non-sens informatiques. Prenons par exemple la condition " Toto est compris entre 5 et 8 ". On peut être tenté de la traduire par : 5 < Toto < 8. Or, une telle expression, qui a du sens en mathématiques, ne veut rien dire en programmation. On va voir tout de suite après comment traduire une telle condition.

4. Conditions composées

Certains problèmes exigent parfois de formuler des conditions qui ne peuvent pas être exprimées sous la forme simple exposée ci-dessus. Reprenons le cas " Toto est inclus entre 5 et 8 ". En fait cette phrase cache non une, mais **deux** conditions. Car elle revient à dire que " Toto est supérieur à 5 et Toto est inférieur à 8 ". Il y a donc bien là deux conditions, reliées par ce qu'on appelle un **opérateur logique**, le mot ET.

Comme on l'a évoqué plus haut, l'informatique met à notre disposition trois opérateurs logiques : ET, OU, et NON.

Le ET a le même sens en informatique que dans le langage courant. Pour que :

> Condition1 ET Condition2 soit VRAI, il faut impérativement que Condition1 soit VRAIE et que Condition2 soit VRAIE.

Il faut se méfier un peu plus du OU. Pour que :

Condition1 OU Condition2 soit VRAI

Il suffit que Condition1 soit VRAIE ou que Condition2 soit VRAIE. Le point important est que si Condition1 est VRAIE et que Condition2 est VRAIE aussi, Condition1 OU Condition2 est VRAIE. Le OU informatique ne veut donc pas dire " ou bien " (sauf dans certaines formes rares, dénommées OU exclusif et notées XOR).

On représente fréquemment tout ceci dans des **tables de vérité** :

C1 ET C2		C1	
		VRAI	FAUX
C2	VRAI	VRAI	FAUX
	FAUX	FAUX	FAUX

C1 OU C2		C1	
		VRAI	FAUX
C2	VRAI	VRAI	VRAI
	FAUX	VRAI	FAUX

C1 XOR C2		C1	
		VRAI	FAUX
C2	VRAI	FAUX	VRAI
	FAUX	VRAI	FAUX

Le gag de la journée : c'est bien sûr formuler une condition qui ne pourra jamais être vraie. Si c'est pas fait exprès, c'est assez rigolo. Si c'est fait exprès, c'est encore plus drôle, car une condition dont on sait d'avance qu'elle sera toujours fausse n'est pas une condition. Cela peut être par exemple : Si Toto < 10 ET Toto > 15 Alors… Bon, ça, c'est un motif immédiat pour payer une tournée générale, et je sens qu'on ne restera pas longtemps le gosier sec.

Quelques mots pour finir là-dessus. L'opérateur NON, lui, inverse une condition :

Condition VRAI ⇔ NON (Condition) FAUX

NON (X > 15) revient à écrire X =< 15

5. Tests imbriqués

Graphiquement, on peut très facilement représenter un SI comme un aiguillage de chemin de fer (ou un aiguillage de train électrique, c'est moins lourd à porter). Un SI ouvre donc deux voies, correspondant à deux traitements différents. Mais il y a des tas de situations où deux voies ne suffisent pas. Par exemple, un programme devant donner l'état de l'eau selon sa température doit pouvoir choisir entre trois réponses possibles (solide, liquide ou gazeuse).

Exemple

```
Variable Temp en Entier
Début
Ecrire "Entrez la température de l'eau :"
Lire Temp
Si Temp =< 0 Alors
Ecrire "C'est de la glace"
Finsi
Si Temp > 0 Et Temp < 100 Alors
Ecrire "C'est du liquide"
Finsi
Si Temp > 100 Alors
Ecrire "C'est de la vapeur"
Finsi
Fin
```

Vous constaterez que c'est un peu laborieux. Les conditions se ressemblent plus ou moins, et surtout on oblige la machine à examiner trois tests successifs alors que tous portent sur une même chose, la température (la valeur de la variable Temp). Il serait ainsi bien plus rationnel d'**imbriquer** les tests de cette manière :

Exemple

```
Variable Temp en Entier
Début
Ecrire "Entrez la température de l'eau :"
Lire Temp
Si Temp =< 0 Alors
Ecrire "C'est de la glace"
Sinon
Si Temp < 100 Alors
Ecrire "C'est du liquide"
Sinon
Ecrire "C'est de la vapeur"
Finsi
Finsi
Fin
```

Nous avons fait des économies au niveau de la frappe du programme : au lieu de devoir taper trois conditions, dont une composée, nous n'avons plus que deux conditions simples. Mais aussi, et surtout, nous avons fait des économies sur le temps d'exécution de l'ordinateur. Si la température est inférieure à zéro, celui-ci écrit dorénavant " C'est de la glace " et passe **directement** à la fin, sans être ralenti par l'examen d'autres possibilités (qui sont forcément fausses).

Cette deuxième version n'est donc pas seulement plus simple à écrire et plus lisible, elle est également plus performante à l'exécution.

Les structures de tests imbriqués sont donc un outil indispensable à la simplification et à l'optimisation des algorithmes.

6. Variables Booléennes

Jusqu'ici, nous avons utilisé uniquement des conditions pour faire des choix. Mais vous vous rappelez qu'il existe un type de variables (les booléennes) susceptibles de stocker les valeurs VRAI ou FAUX. En fait, on peut donc entrer des conditions dans ces variables, et tester ensuite la valeur de ces variables.

Reprenons l'exemple de l'eau. On peut le réécrire ainsi :

Exemple

```
Variable Temp en Entier
Variables A, B en Booléen
Début
Ecrire "Entrez la température de l'eau :"
Lire Temp
A ⇦ Temp =< 0
B ⇦ Temp < 100
Si A Alors
Ecrire "C'est de la glace"
Sinon
B Alors
Ecrire "C'est du liquide"
Sinon
Ecrire "C'est de la vapeur"
Finsi
Finsi
Fin
```

A priori, cette technique ne présente guère d'intérêt : on a alourdi plutôt qu'allégé l'algorithme de départ, en lui faisant recourir à deux variables supplémentaires. Mais :

- une variable booléenne n'a besoin que d'un seul bit pour être stockée. L'alourdissement de ce point de vue n'est donc pas considérable.
- dans certains cas, notamment celui de conditions composées très lourdes (avec plein de ET et de OU tout partout) cette technique peut faciliter le travail du programmeur. Les variables booléennes peuvent

également s'avérer très utiles pour servir de **flag**, technique dont on reparlera plus loin (rassurez-vous, rien à voir avec le flagrant délit des policiers).

7. Quelques jeux logiques

Une remarque pour commencer : dans le cas de conditions composées, les parenthèses jouent un rôle important.

```
Variables A, B, C, D, E en Booléen
Variable X en Entier
Début
Lire X
A ⇦ X < 2
B ⇦ X > 12
C ⇦ X < 6
D ⇦ (A ET B) OU C
E ⇦ A ET (B OU C)
Ecrire D, E
Fin
```

Si X = 3, alors on remarque que D sera VRAI alors que E sera FAUX.

S'il n'y a que des ET, ou que des OU, en revanche, les parenthèses ne changent strictement rien.

On en arrive à une autre propriété des ET et des OU, bien plus intéressante. Spontanément, on pense souvent que ET et OU s'excluent mutuellement, et qu'un problème donné s'exprime soit avec un ET, soit avec un OU. Pourtant, ce n'est pas si évident.

Quand faut-il ouvrir la fenêtre de la salle ? Uniquement si les conditions l'imposent, à savoir :

```
Si il fait trop chaud ET il ne pleut pas Alors
Ouvrir la fenêtre
Sinon
Fermer la fenêtre
Finsi
```

Cette petite règle pourrait tout autant être formulée comme suit :

```
Si il ne fait pas trop chaud OU il pleut Alors
Fermer la fenêtre
Sinon
Ouvrir la fenêtre
Finsi
```

Ces deux formulations sont strictement équivalentes. Ce qui nous amène à la conclusion suivante : toute structure de test requérant une condition composée faisant intervenir l'opérateur ET peut être exprimée de manière équivalente avec un opérateur OU, et réciproquement.

Ceci est moins surprenant qu'il n'y paraît au premier abord. Jetez pour vous en convaincre un œil sur les tables de vérité, et vous noterez la symétrie entre celle du ET et celle du OU.
Bien sûr, on ne peut pas se contenter de remplacer purement et simplement les ET par des OU ; ce serait un peu facile. La règle d'équivalence est la suivante (on peut la vérifier sur l'exemple de la fenêtre) :

```
Si A ET B Alors Si NON A OU NON B Alors
Instructions 1 Instructions 2
Sinon ⇔Sinon
Instructions 2 Instructions 1
Finsi Finsi
```

IV Les boucles

Et ça y est, on y est, on est arrivés, la voilà, c'est Broadway, la quatrième et dernière structure : ça est les **boucles**. Si vous voulez épater vos amis, vous pouvez également parler de **structures répétitives**, voire carrément de **structures itératives**. Ca calme, hein ? Bon, vous faites ce que vous voulez, ici on est entre nous, on parlera de boucles.

1. A quoi cela sert-il donc ?

Prenons le cas d'une saisie au clavier (une lecture), par exemple, on pose une question à laquelle l'utilisateur doit répondre par O (Oui) ou N (Non). Mais l'utilisateur, facétieux ou maladroit, risque fort tôt ou tard de taper autre chose que cela. Dès lors, le programme peut soit planter par une erreur d'exécution (parce que le type de réponse ne correspond pas au type de la variable attendu) soit se dérouler normalement jusqu'au bout, mais en produisant des résultats fantaisistes.

Alors, dans tout programme un tant soit peu sérieux, on met en place ce qu'on appelle un **contrôle de saisie** (pour vérifier que les données entrées au clavier correspondent bien à celles attendues par l'algorithme.

A vue de nez, on pourrait essayer avec un SI. Voyons voir ce que ça donne :

```
Variable Rep en Caractère
Ecrire "Voulez vous un café ? (O/N)"
Lire Rep
Si Rep <> "O" ET Rep <> "N" Alors
Ecrire "Saisie erronnée. Recommencez"
Lire Rep
FinSi
```

C'est impeccable. Du moins tant que l'utilisateur a le bon goût de ne se tromper qu'une seule fois, et d'entrer une valeur correcte à la deuxième demande. Si l'on veut également bétonner en cas de deuxième erreur, il faudrait rajouter un SI. Et ainsi de suite, on peut rajouter des centaines de SI, et écrire un algorithme aussi lourd qu'une blague des Grosses Têtes, on n'en sortira pas, il y aura toujours moyen qu'un acharné flanque le programme par terre. C'est donc une impasse.

2. Une première structure de boucle

La seule issue est donc cette structure de boucle, qui se présente ainsi :

```
TantQue booléen
...
Instructions
...
FinTantQue
```

Le principe est simple : le programme arrive sur la ligne du TantQue. Il examine alors la valeur du booléen (qui, je le rappelle, peut être une variable booléenne ou, plus fréquemment, une condition). Si cette valeur est VRAI, le programme exécute les instructions qui suivent, jusqu'à ce qu'il rencontre la ligne FinTantQue. Il retourne ensuite sur la ligne du TantQue, procède au même examen, et ainsi de suite. Le manège enchanté ne s'arrête que lorsque le booléen prend la valeur FAUX.

Illustration avec notre problème de contrôle de saisie :

```
Variable Rep en Caractère
Ecrire "Voulez vous un café ? (O/N)"
TantQue Rep <> "O" ET Rep <> "N"
Lire Rep
```

```
Si Rep <> "O" ET Rep <> "N" Alors
Ecrire "Saisie erronnée. Recommencez"
FinSi
FinTantQue
```

On remarquera que la présence du bloc " Si " est uniquement là pour l'affichage éventuel du libellé de saisie erronée. En lui-même, le bloc " tant que " est d'une simplicité biblique.

Le gag de la journée : c'est d'écrire une structure TantQue dans laquelle le booléen ne devient jamais faux ! L'ordinateur tourne alors dans la boucle comme un dératé et n'en sort plus. Seule solution, quitter le programme avec un démonte-pneu ou un bâton de dynamite.

Cette faute de programmation grossière – mais fréquente - ne manquera pas d'égayer l'ambiance collective de cette formation… et accessoirement d'étancher la soif proverbiale de vos enseignants.

3. Boucler en comptant, ou compter en bouclant

Dans le dernier algorithme, vous avez remarqué qu'une boucle était fréquemment utilisée pour augmenter la valeur d'une variable. Il arrive également très souvent qu'on ait besoin d'effectuer un nombre déterminé de passages. Or, a priori, notre structure TantQue ne sait pas à l'avance combien de tours de boucle elle va effectuer (cela dépend d'une condition).

C'est pourquoi une autre structure de boucle est à notre disposition :

```
Variable Truc en Entier
Truc ⇦ 0
TantQue Truc < 15
Truc = Truc + 1
Ecrire "Passage numéro : ", Truc
FinTantQue
```

Equivaut à :

```
Variable Truc en Entier
Pour Truc = 1 à 15
Ecrire "Passage numéro : ", Truc
Truc Suivant
```

Au sens strict, on pourrait donc s'en dispenser, mais c'est tellement agréable de faire moins d'efforts…

4. Des boucles dans des boucles

On rigole, on rigole !

De même que les poupées russes contiennent d'autres poupées russes, de même qu'une structure SI … ALORS peut contenir d'autres structures SI … ALORS, une boucle peut contenir d'autres boucles.

```
Variables Truc, Trac en Entier
Pour Truc ⇦ 1 à 15
Ecrire "Il est passé par ici"
Pour Trac ⇦ 1 à 6
Ecrire "Il repassera par là"
Trac Suivant
Truc Suivant
```

Dans cet exemple, le programme écrira une fois "il est passé par ici" puis six fois de suite "il repassera par là", et ceci quinze fois en tout. A la fin, il y aura donc eu 15 x 6 = 90 passages dans la deuxième boucle (celle du milieu).

Notez la différence marquante avec cette structure :

```
Variables Truc, Trac en Entier
Pour Truc ⇦ 1 à 15
Ecrire "Il est passé par ici"
Truc Suivant
Pour Trac ⇦ 1 à 6
Ecrire "Il repassera par là"
Trac Suivant
```

Ici, il y aura quinze écritures consécutives de "il est passé par ici", puis six écritures consécutives de "il repassera par là" , et ce sera tout.

Si des boucles peuvent être imbriquées (cas n°1) ou successives (cas n°2), elles ne peuvent jamais, au grand jamais, être croisées. Cela n'aurait aucun sens logique, et de plus, bien peu de langages vous autoriseraient ne serait-ce qu'à écrire cette structure aberrante.

```
Variables Truc, Trac en Entier


Pour Truc ⇦...
instructions
Pour Trac ⇦ ...
instructions
Truc Suivant

instructions
Trac Suivant
```

5. Et encore une tournée générale !

Examinons l'algorithme suivant :

```
Variable Truc en Entier
Pour Truc ⇦ 1 à 15
Truc ⇦ Truc * 2
Ecrire "Passage numéro : ", Truc
Truc Suivant
```

Vous remarquerez que nous faisons ici gérer " en double " la variable Truc, ces deux gestions étant contradictoires. D'une part, la ligne " Pour… " augmente la valeur de Truc de 1 à chaque passage. D'autre part la ligne " Truc ⇦ Truc * 2 " double la valeur de Truc à chaque passage. Il va sans dire que de telles manipulations perturbent complètement le déroulement normal de la boucle, et sont sources d'exécutions erratiques.

Le gag de la journée : consiste donc à manipuler, au sein d'une boucle Pour, la variable qui sert de compteur à cette boucle. Cette technique est à proscrire absolument… sauf bien sûr, si vous cherchez un prétexte pour régaler tout le monde au bistrot.

Mais dans ce cas, n'ayez aucune inhibition, proposez-le directement, pas besoin de prétexte.

V Les tableaux

Bonne nouvelle, on a vu toutes les structures logiques de la programmation.

Mauvaise nouvelle, il vous reste tout de même quelques petites choses à apprendre.

1. Utilité des tableaux

Imaginons que dans un programme, nous ayons besoin simultanément de 12 valeurs (par exemple, des notes pour calculer une moyenne). Evidemment, la seule solution dont nous disposons à l'heure actuelle consiste à déclarer quinze variables, appelées par exemple Notea, Noteb, Notec, etc. Bien sûr, on peut opter pour une notation un peu simplifiée, par exemple N1, N2, N3, etc. Mais cela ne change pas fondamentalement notre problème, car arrivé au calcul, cela donnera obligatoirement une atrocité du genre :

Moy = (N1 + N2 + N3 + N4 + N5 + N6 + N7 + N8 + N9 + N10 + N11 + N12) / 12

Ouf ! C'est tout de même bigrement laborieux. Et pour un peu que nous soyons dans un programme de gestion avec quelques centaines ou quelques milliers de valeurs à traiter, alors là c'est direct le suicide.

Cerise sur le gâteau, pour peu que l'on ne puisse savoir d'avance combien il y aura de valeurs à traiter, là on est carrément cuits.

C'est pourquoi la programmation nous permet de rassembler toutes ces variables en une seule, au sein de laquelle chaque valeur sera désignée par un numéro. En bon français, cela donnerait donc quelque chose du genre " la note numéro 1 ", " la note numéro 2 ", " la note numéro 8 ". C'est largement plus pratique, vous vous en doutez.

Un ensemble de valeurs portant ainsi le même nom de variable et repérées par un nombre, s'appelle un **tableau**, ou encore une **variable indicée**. Et le nombre qui sert à repérer chaque valeur s'appelle – ô surprise – un **indice**.

2. Notation et utilisation algorithmique.

Dans notre exemple, nous créerons donc un tableau appelé Note (ou N, si on veut aller plus vite). Et chaque note individuelle sera désignée Note(1), Note(2), etc.

Un tableau doit être déclaré comme tel, en précisant le nombre et le type de valeurs qu'il contiendra.

Tableau Note(12) **en Entier**

On peut créer des tableaux contenant des variables de tous types : tableaux de numériques, bien sûr, mais aussi tableaux de caractères, de booléens, de tout ce qui existe dans un langage donné comme type de variables. Par contre, hormis dans quelques rares langages, on ne peut pas faire un mixage de types différents de valeurs au sein d'un même tableau.

L'énorme avantage des tableaux, c'est qu'on va pouvoir les traiter en faisant des boucles. Par exemple, pour effectuer notre calcul de moyenne, cela donnera :

```
Tableau Note(12) en Entier
Variables i, Som en Entier
Variable Moy en Réel
...
Som ⇦ 0
Pour i ⇦ 1 à 12
Som = Som + Note(i)
i Suivant
Moy = Som / 12
```

NB : Cet exemple ne peut être qu'un extrait, dans la mesure où on n'a pas programmé comment s'effectue le remplissage des valeurs du tableau.

L'indice qui sert à repérer les éléments du tableau peut être exprimé directement comme un nombre en clair, mais il peut être aussi une variable, ou une expression calculée.

La valeur d'un indice doit toujours :

- **être égale au moins à 0 ou à 1** (dans certains langages, le premier élément d'un tableau porte l'indice zéro, dans d'autres l'indice 1)
- **être un nombre entier**. Quel que soit le langage, l'élément Truc(3,1416) n'existe jamais
- **être inférieure ou égale au nombre d'éléments du tableau**. Si le tableau Bidule a été déclaré comme ayant 25 éléments, la présence dans une ligne, sous une forme ou sous une autre, de Bidule(26) déclenchera automatiquement une erreur.

Le gag de la journée : consiste à confondre, dans sa tête et / ou dans un algorithme, **l'indice** d'un élément d'un tableau avec son **contenu**. La troisième maison de la rue n'a pas forcément trois habitants, et la vingtième vingt habitants. En notation algorithmique, il n'y a aucun rapport entre i et truc(i).

3. Tableaux dynamiques

Il arrive fréquemment que l'on ne connaisse pas à l'avance le nombre d'éléments que devra comporter le tableau. Bien sûr, une solution consisterait à déclarer un tableau gigantesque (10 000 éléments, pourquoi pas, au diable les varices) pour être sûr que " ça rentre ". Mais d'une part, on n'en sera jamais parfaitement sûr, d'autre part, en raison de l'immensité de la place mémoire réservée – et la plupart du temps non utilisée, c'est un gâchis préjudiciable à la rapidité de notre algorithme.

Aussi, pour parer à ce genre de situation, a-t-on la possibilité de déclarer le tableau sans préciser son nombre d'éléments, puis au cours du programme, à préciser ce nombre via l'instruction **Redim**. Notez que **tant qu'on n'a pas précisé le nombre d'éléments d'un tableau, d'une manière ou d'une autre, ce tableau est inutilisable**.

Exemple : on veut faire saisir des notes pour un calcul de moyenne, mais on ne sait pas combien il y aura de notes à saisir. Le début de l'algorithme sera quelque chose du genre :

```
Tableau Notes() en Entier
Variable nb en Entier
Début
Ecrire "Combien y a-t-il de notes à saisir ?"
Lire nb
Redim Notes(nb)
...
```

4. Tri d'un tableau

Ce qui suit est incontournable. Combien de fois au cours d'une carrière (brillante) de développeur a-t-on besoin de ranger des valeurs dans un ordre donné ? C'est incalculable. Aussi, plutôt qu'avoir à réinventer à chaque fois la roue, le fusil à tirer dans les coins, le fil à couper le roquefort et la poudre à maquiller, vaut-il mieux avoir assimiler quelques techniques solidement éprouvées, même si elles paraissent un peu ardues au départ.

Il existe plusieurs stratégies possibles pour trier les éléments d'un tableau ; nous en verrons une : le tri par sélection.

Admettons que le but de la manœuvre soit de trier un tableau de 12 éléments dans l'ordre croissant.

45	122	12	3	21	78	64	53	89	28	84	46

On commence par rechercher le plus petit élément. On l'identifie en quatrième position, et on l'échange alors avec l'élément numéro 1.

3	122	12	45	21	78	64	53	89	28	84	46

On recommence à rechercher le plus petit élément, mais cette fois à partir du deuxième élément. On le trouve en 3e position, on échange donc le deuxième avec le troisième :

3	12	122	45	21	78	64	53	89	28	84	46

On recommence à partir du troisième, ce qui donnera in fine :

3	12	21	45	122	78	64	53	89	28	84	46

Et cetera, et cetera, jusqu'à l'avant dernier.

L'algorithme permettant d'effectuer cette tâche est le suivant :

```
Pour i ⇦ 1 à 11
mini = t(i)
posmini = i
Pour j ⇦ i + 1 à 12
Si t(j) < mini alors
mini ⇦ t(j)
posmini ⇦ j
Finsi
j suivant
t(posmini) ⇦ t(i)
t(i) ⇦ mini
i suivant
```

VI Tableaux multidimensionnels

Ceci n'est pas un dérèglement de votre téléviseur. Nous contrôlons tout, nous savons tout, et les phénomènes paranormaux que vous constatez sont dus au fait que vous êtes passés dans… la quatrième dimension (musique angoissante : " tintintin… ").

Oui, enfin bon, avant d'attaquer la quatrième, on va déjà se coltiner la deuxième.

1. Pourquoi plusieurs dimensions ?

Une seule ne suffisait-elle pas déjà amplement à notre bonheur, me demanderez-vous ? Certes, répondrai-je, mais vous allez voir qu'avec deux (et davantage encore) c'est carrément le nirvana.

Prenons le cas de la modélisation d'un jeu de dames, et du déplacement des pions sur le damier. Je rappelle qu'un pion qui est sur une case blanche peut se déplacer (pour simplifier) sur les quatre cases blanches adjacentes.

Avec les outils dont on dispose actuellement, le plus simple serait évidemment de modéliser le damier sous la forme d'un tableau. Chaque case est un emplacement du tableau, qui contient par exemple 0 si elle est vide, et 1 s'il y a un pion. On attribue comme indices aux cases les numéros 1 à 8 pour la première ligne, 9 à 16 pour la deuxième ligne, et ainsi de suite jusqu'à 64.

Un pion placé dans la case numéro i, autrement dit la valeur 1 de Cases(i), peut bouger vers les cases contiguës en diagonale. Cela va nous obliger à de petites acrobaties intellectuelles : la case située juste au-dessus de la case numéro i ayant comme indice i-8, les cases valables sont celles d'indice i-7 et i-9. De même, la case située juste en dessous ayant comme indice i+8, les cases valables sont celles d'indice i+7 et i+9.

Bien sûr, on peut fabriquer tout un programme comme cela, mais le moins qu'on puisse dire est que cela ne facilite pas la clarté de l'algorithme.

Il serait évidemment plus simple de modéliser un damier par… un damier !

2. Tableaux à deux dimensions

L'informatique nous offre la possibilité de déclarer des tableaux dans lesquels les valeurs ne sont pas repérées par une seule, mais par deux coordonnées.

Un tel tableau se déclare ainsi :

Tableau Cases(8, 8) **en Entier**

Cela veut dire : réserve moi un espace de mémoire pour 8 x 8 entiers, et quand j'aurai besoin de l'une de ces valeurs, je les repèrerai par deux indices (comme à la bataille navale, ou Excel, la seule différence étant que pour les coordonnées, on n'utilise pas de lettres, juste des chiffres).

Pour notre problème de dames, les choses vont sérieusement s'éclaircir. La case qui contient le pion est dorénavant Cases(i , j). Et les quatre cases disponibles sont Cases(i-1, j-1), Cases(i-1, j+1), Cases(i+1, j-1) et Cases(i+1, j+1).

Une remarque : il n'y a aucune différence qualitative entre un tableau à deux dimensions (i, j) et un tableau à une dimension (i * j). De même que le jeu de dames qu'on vient d'évoquer, tout problème pouvant être modélisé d'une manière peut aussi être modélisé de l'autre. Simplement, l'une ou l'autre de ces techniques correspond plus spontanément à tel ou tel problème, et facilite donc (ou complique, si on a choisi la mauvaise option) l'écriture et la lisibilité de l'algorithme.

Une autre remarque : une question classique à propos des tableaux à deux dimensions est de savoir si le premier indice représente les lignes ou le deuxième les colonnes, ou l'inverse. Je ne répondrai pas à cette question non parce que j'ai décidé de bouder, mais **parce qu'elle n'a aucun sens**. " Lignes " et " Colonnes " sont des

concepts graphiques, qui s'appliquent à des objets du monde réel ; les indices des tableaux ne sont que des coordonnées logiques, pointant sur des adresses de mémoire vive. Si cela ne vous convainc pas, pensez à un jeu de bataille navale classique : les lettres doivent-elles désigner les lignes et les chiffres les colonnes ? Aucune importance ! Chaque joueur peut même choisir une convention différente, aucune importance ! L'essentiel est qu'une fois une convention choisie, un joueur conserve la même tout au long de la partie, bien entendu.

3. Tableaux à n dimensions

Si vous avez compris le principe des tableaux à deux dimensions, sur le fond, il n'y a aucun problème à passer au maniement de tableaux à trois, quatre, ou pourquoi pas neuf dimensions. C'est exactement la même chose. Si je déclare un tableau Titi(3, 5, 4, 4), il s'agit d'un espace mémoire contenant 3 x 5 x 4 x 4 = 240 valeurs. Chaque valeur est repérée par quatre coordonnées.

Le principal obstacle au maniement systématique de ces tableaux à plus de trois dimensions est que le programmeur, quand il conçoit son algorithme, aime bien faire des petits gribouillis, des dessins immondes, imaginer les boucles dans sa tête, etc. Or, autant il est facile d'imaginer concrètement un tableau à une dimension, autant cela reste faisable pour deux dimensions, autant cela devient l'apanage d'une minorité privilégiée pour les tableaux à trois dimensions (je n'en fais malheureusement pas partie) et hors de portée de tout mortel au-delà. C'est comme ça, l'esprit humain a du mal à se représenter les choses dans l'espace, et crie grâce dès qu'on est dans l'hyperespace (oui, c'est comme ça que ça s'appelle au delà de trois dimensions).

Donc, pour des raisons uniquement pratiques, les tableaux à plus de trois dimensions sont rarement utilisés par des programmeurs non matheux (car les matheux, de par leur formation, ont une fâcheuse propension à manier des espaces à n dimensions comme qui rigole, mais ce sont bien les seuls, et laissons les dans leur coin, c'est pas des gens comme nous).

VII Les fonctions prédéfinies

Certains traitements ne peuvent être effectués par un algorithme, aussi savant soit-il.

C'est par exemple le cas du calcul du sinus d'un angle : pour en obtenir une valeur approchée, il faudrait appliquer une formule d'une complexité à vous glacer le sang. Aussi, que se passe-t-il sur les petites calculatrices que vous connaissez tous ? On vous fournit quelques touches spéciales, dites **touches de fonction**s, qui vous permettent par exemple de connaître immédiatement ce résultat. Sur votre calculatrice, si vous voulez connaître le sinus de 35°, vous taperez 35, puis la touche SIN, et vous aurez le résultat.

Tout langage de programmation propose ainsi un certain nombre de **fonctions** ; certaines sont indispensables, car elles permettent d'effectuer des traitements qui seraient sans elles impossibles. D'autres servent à soulager le programmeur, en lui épargnant de longs algorithmes.

1. Structure générale des fonctions

Reprenons l'exemple du sinus. Les langages proposent généralement une fonction SIN. Si nous voulons stocker le sinus de 35 dans la variable A, nous écrirons :

A ⇦ Sin(35)

Une fonction est donc constituée de trois parties :

- le **nom** proprement dit de la fonction. Ce nom ne s'invente pas ! Il doit impérativement correspondre à une fonction proposée par le langage.
- deux parenthèses, une ouvrante, une fermante
- une liste de valeurs, indispensables à la bonne exécution de la fonction. Ces valeurs s'appellent des **arguments**, ou des **paramètres**. Certaines fonctions exigent un seul argument, d'autres deux, etc. et d'autres encore aucun. A noter que même dans le cas de ces fonctions n'exigeant aucun argument, les parenthèses restent obligatoires. Le nombre d'arguments nécessaire pour une fonction donnée ne s'invente pas : il est fixé par le langage. Par exemple, la fonction sinus a besoin d'un argument (logique, c'est la valeur de l'angle). Si vous essayez de l'exécuter en lui donnant deux arguments, ou aucun, cela déclenchera une erreur à l'exécution. Notez également que les arguments doivent être d'un certain **type**, et qu'il faut respecter ces types.

2. Les fonctions de texte

Une catégorie privilégiée de fonctions est celle qui nous permet de manipuler des chaînes de caractères. Nous avons déjà vu qu'on pouvait facilement " coller " deux chaînes l'une à l'autre avec l'opérateur de concaténation &. Mais ce que nous ne pouvions pas faire, et qui va être maintenant possible, c'est pratiquer des extractions de chaînes (moins douloureuses, il faut le noter, que les extractions dentaires).

Tous les langages, je dis bien tous, proposent peu ou prou les fonctions suivantes, même si le nom et la syntaxe peuvent varier d'un langage à l'autre :

Len(chaîne)	renvoie le nombre de caractères d'une chaîne
Mid(chaîne, n1, n2)	renvoie un extrait de la chaîne, commençant au caractère n1 et faisant n2 caractères de long.
Left(chaîne, n)	renvoie les n caractères les plus à gauche dans chaîne.
Right(chaîne, n)	renvoie les n caractères les plus à droite dans chaîne

Enfin, souvent on dispose également de :

Trouve(chaîne1, chaîne2)	renvoie un nombre correspondant à la position de chaîne2 dans chaîne1. Si chaîne2 n'est pas comprise dans chaîne1, la fonction renvoie zéro.

Exemples :

Len("Bonjour, ça va ?")	vaut	16
Len("")	vaut	0
Mid("Zorro is back", 4, 6)	vaut	"o est a"
Mid("Zorro is back", 12, 1)	vaut	"r"
Left("Et pourtant…", 8)	vaut	"Et pourt"
Right("Et pourtant…", 4)	vaut	"t…"
Trouve("Un pur bonheur", "pur")	vaut	3
Trouve("Un pur bonheur", "techno")	vaut	0

Il existe aussi dans tous les langages une fonction qui renvoie le caractère correspondant à un code Ascii donné (fonction Asc), et Lycée de Versailles (fonction Chr) :

Asc("N") vaut 78

Chr(63) vaut "?"

Avec tout çà, on va faire des miracles.

3. Deux fonctions classiques

Une fonction extrêmement répandue est celle qui permet de récupérer la partie entière d'un nombre :

Après :A ⇦ Ent(3,228)A vaut 3

Cette fonction est notamment indispensable pour effectuer le célébrissime test de parité (voir exercice dans pas longtemps).

Une autre fonction classique , car très utile, est celle qui génère un nombre choisi au hasard.

Tous les programmes de jeu, ou presque, ont besoin de ce type d'outils, qu'il s'agisse de simuler un lancer de dés ou le déplacement chaotique du vaisseau spatial de l'enfer de la mort piloté par l'infâme Zorglub, qui veut faire main basse sur l'Univers (heureusement vous êtes là pour l'en empêcher, ouf).

Mais il n'y a pas que les jeux qui ont besoin de générer des nombres aléatoires. La modélisation (physique, géographique, économique, etc.) a parfois recours à des modèles dits stochastiques (chouette, encore un nouveau mot savant !). Ce sont des modèles dans lesquels les variables se déduisent les unes des autres par des relations déterministes (autrement dit des calculs), mais où l'on simule la part d'incertitude par une " fourchette " de hasard.

Par exemple, un modèle démographique supposera qu'une femme a en moyenne x enfants au cours de sa vie, mettons 1,5. Mais il supposera aussi que sur une population donnée, ce chiffre peut fluctuer entre 1,35 et 1,65 (si on laisse une part d'incertitude de 10%). Chaque année, c'est-à-dire chaque série de calcul des valeurs du modèle, on aura ainsi besoin de faire choisir à la machine un nombre au hasard compris entre 1,35 et 1,65.

Dans tous les langages, cette fonction existe et produit le résultat suivant :

Après :Toto ⇦ Alea()0 =< Toto < 1

Il est indispensable d'apprendre à manier cette fonction, alors autant que ce soit fait.

VIII Les fichiers séquentiels

Jusqu'à présent, les informations utilisées dans nos programmes ne pouvaient provenir que de deux sources : soit elles étaient inclues dans l'algorithme lui-même, par le programmeur, soit elles étaient entrées en cours de route par l'utilisateur. Mais évidemment, cela ne suffit pas aux besoins réels.

Imaginons que l'on veuille écrire un programme gérant un carnet d'adresses. D'une exécution à l'autre, l'utilisateur doit pouvoir retrouver son carnet à jour, avec les modifications qu'il y a apportées la dernière fois qu'il a exécuté le programme. Les données du carnet d'adresse ne peuvent donc être inclues dans l'algorithme, et encore moins être entrées au clavier à chaque nouvelle exécution !

Les fichiers sont là pour combler ce manque. Ils servent à stocker des informations de manière permanente, entre deux exécutions d'un programme. Car si les variables, qui sont je le rappelle des adresses de mémoire vive, disparaissent à chaque fin d'exécution, les fichiers, eux sont stockés sur des périphériques à mémoire de masse (disquette, disque dur, CD Rom…).

1. Organisation des fichiers texte

Il existe deux grandes variantes pour structurer les données au sein d'un fichier texte : la délimitation, ou les champs de largeur fixe. Reprenons le cas du carnet d'adresse ; Nous nous limiterons aux renseignements suivants : Nom, prénom, téléphone, e-mail. Les données, sur le fichier texte, peuvent être organisées ainsi :

Structure n°1

"Fonfec";"Sophie";0142156487;"fonfec@yahoo.fr"
"Zétofrais";"Mélanie";0456912347 ;"zétofrais@free.fr"
"Herbien";"Jean-Philippe";0289765194;"vantard@free.fr"
"Hergébel";"Octave";0149875231;"rg@aol.fr"

ou ainsi :

Structure n°2

Fonfec	Sophie	0142156487fonfec@yahoo.fr
Zétofrais	Mélanie	0456912347zétofrais@free.fr
Herbien	Jean-Philippe	0289765194vantard@free.fr
Hergébel	Octave	0149875231rg@aol.fr

La structure n°1 est dite **délimitée** ; Elle utilise un **caractère de délimitation**, qui permet de repérer quand commence un champ et quand commence le suivant. Il va de soi que ce caractère de délimitation doit être strictement interdit à l'intérieur de chaque champ, faute de quoi la structure devient proprement illisible.

La structure n°2, elle, est dite à **champs de largeur fixe**. Il n'y a pas de caractère de délimitation, mais on sait que les x premiers caractères de chaque ligne stockent le nom, les y suivants le prénom, etc. Cela impose bien entendu de ne pas saisir un renseignement plus long que le champ prévu pour l'accueillir.

- L'avantage de la structure n°1 est son **faible encombrement en place mémoire** ; il n'y a aucun espace perdu, et un fichier texte codé de cette manière occupe le minimum de place possible. Mais elle possède en revanche un inconvénient majeur, qui est la **lenteur de la lecture**. En effet, chaque fois que l'on récupère une ligne dans le fichier, il faut alors parcourir un par un tous les caractères pour repérer chaque occurrence du caractères de séparation avant de pouvoir découper cette ligne en différents champs.
- La structure n°2, à l'inverse, **gaspille de la place mémoire**, puisque le fichier est un vrai gruyère plein de trous. Mais d'un autre côté, la récupération des différents champs est très **rapide**. Lorsqu'on récupère une ligne, il suffit de la découper en différentes chaînes de longueur prédéfinie, et le tour est joué.

A l'époque où la place mémoire coûtait cher, la structure délimitée était privilégiée. Mais depuis bien des années, la quasi-totalité des logiciels – et des programmeurs – optent pour la structure en champs de largeur fixe. Aussi, sauf mention contraire, nous ne travaillerons qu'avec des fichiers bâtis sur cette structure.

2. Types d'accès

On vient de voir que l'organisation des données au sein du fichier pouvait s'effecteur selon deux grands choix stratégiques. Mais il existe une autre ligne de partage des fichiers : le **type d'accès**, autrement dit la manière dont la machine va pouvoir aller rechercher les enregistrements.

On distingue :

- **L'accès séquentiel** : on ne peut accéder qu'à l'enregistrement suivant celui qu'on vient de lire.
- **L'accès direct** (ou **aléatoire**) : on peut accéder directement à l'enregistrement de son choix, en précisant le numéro de cet enregistrement.

A la différence de la précédente, cette typologie ne se reflète pas dans la structure elle-même du fichier. En fait, tout fichier, quelle que soit sa structure (délimité ou en largeur fixe) peut être utilisé avec l'un ou l'autre des deux grands types d'accès. Le choix du type d'accès n'est pas un choix qui concerne le fichier lui-même, mais uniquement la manière dont il va être traité par la machine. C'est donc dans le programme, et seulement dans le programme, que l'on choisit le type d'accès souhaité.

Au premier abord, il est clair que l'accès direct possède tous les avantages et aucun inconvénient. Sa programmation est immensément plus simple et moins fastidieuse que celle de l'accès séquentiel. Mais… vous vous doutez bien qu'il y a un mais.

Le gros problème de l'accès direct, c'est qu'il n'est pas géré de la même manière selon le type de machine. Car la gestion de l'accès direct suppose des éléments de langage " annexes ", qui diffèrent selon les ordinateurs et qui ne seront pas forcément présents. Pour résumer, disons que si vous écrivez un programme C utilisant des accès directs à des fichiers, vous n'êtes pas absolument certains que votre programme pourra tourner sur tous les ordinateurs.

En revanche, l'accès séquentiel, lui, est entièrement piloté par le langage lui-même. Il est donc laborieux à programmer, rustique, tout ce que vous voulez, mais il offre une sécurité absolue de portabilité. Voilà pourquoi la plupart des programmeurs s'astreignent encore à utiliser ce type d'accès, et voilà pourquoi c'est celui-là que nous allons étudier.

Et de toute manière, qui peut le plus peut le moins, et une fois que vous aurez maîtrisé la programmation de l'accès séquentiel, celle d l'accès direct vous apparaîtra comme une ineffable poilade.

3. Instructions

Si l'on veut travailler sur un fichier texte, la première chose à faire est de l'ouvrir. Cela se fait en attribuant au fichier un numéro de canal. On ne peut ouvrir qu'un seul fichier par canal, mais on dispose toujours de plusieurs canaux, dont pas de soucis.

L'important est que lorsqu'on ouvre un fichier, on stipule ce qu'on va en faire : lire, écrire ou ajouter.

- Si on ouvre un fichier **pour lecture**, on ne pourra que récupérer les informations qu'il contient, sans les modifier en aucune manière.
- Si on ouvre un fichier **pour écriture**, on pourra mettre dedans toutes les informations que l'on veut. Mais les informations précédentes, si elles existent, seront intégralement écrasées.
- Si on ouvre un fichier **pour ajout**, on ne peut ni lire, ni modifier les infos existantes. Mais on pourra, comme vous commencez à vous en douter, ajouter de nouveaux enregistrements.

Au premier abord, ces limitations peuvent sembler infernales. Mais avec un peu d'habitude, on se rend compte qu'en fait on peut faire tout ce qu'on veut avec ces fichiers séquentiels.

Pour ouvrir un fichier texte, on écrira par exemple :

Ouvrir "Exemple.txt" sur 4 **pour Lecture**

Ici, "Exemple.txt" est le nom du fichier sur le disque dur, 4 est le numéro de canal, et ce fichier a donc été ouvert en lecture, Christine, j'crois qu'c'est clair.

```
Ouvrir "Exemple.txt" sur 4 en Lecture
Variables Truc, Nom, Prénom, Tel, Mail en Caractères

Lire 4, Truc
Nom ⇦ Mid(Truc, 1, 20)
Prénom ⇦ Mid(Truc, 21, 15)
Tel ⇦ Mid(Truc, 36, 10)
Mail ⇦ Mid(Truc, 46, 20)
```

L'instruction Lire récupère donc dans la variable spécifiée l'enregistrement suivant dans un fichier. Suivant par rapport à quoi ? Par rapport au dernier enregistrement lu. C'est en cela que le fichier est dit séquentiel.

Lire un fichier séquentiel de bout en bout suppose donc de programmer une boucle. Comme on sait rarement combien d'enregistrements comporte le fichier, la combine consiste à utiliser la fonction EOF (acronyme pour End Of File). Cette fonction est vraie si on a atteint la fin du fichier (auquel cas une lecture supplémentaire déclencherait une erreur. L'algorithme ultra classique en pareil cas est donc :

```
Variable Truc en Caractère
Ouvrir "Exemple.txt" sur 5 en Lecture
Tantque Non EOF(5)
Lire 5, Truc
...
FinTantQue
```

Pour une opération d'écriture, ou d'ajout, il faut d'abord constituer une chaîne équivalente à la nouvelle ligne du fichier. Cette chaîne doit donc être " calibrée " de la bonne manière, avec les différents champs qui " tombent " aux emplacements corrects. Le moyen le plus simple pour s'épargner de longs traitements est de procéder avec des chaînes correctement dimensionnées dès leur déclaration (la plupart des langages offrent cette possibilité) :

```
Ouvrir "Exemple.txt" sur 3 en Ajout
Variable Truc en Caractère
Variables Nom*20, Prénom*15, Tel*10, Mail*15 en Caractère
…
Nom ⇦ "Joker"
Prénom ⇦ "Midnight"
Tel ⇦ "0348946532"
Mail ⇦ "allstars@rockandroll.com"
Truc ⇦ Nom & Prénom & Tel & Mail
Ecrire 3, Truc
```

Et pour finir, une fois qu'on en a terminé avec un fichier, il ne faut pas oublier de fermer ce fichier. On libère ainsi le canal qu'il occupait (et accessoirement, on pourra utiliser ce canal dans la suite du programme pour un autre fichier… ou pour le même).

4. Stratégies de traitement

Il existe globalement deux manières de traiter les fichiers textes :

- l'une consiste à s'en tenir au fichier proprement dit. C'est parfois un peu acrobatique, lorsqu'on veut supprimer un élément d'un fichier : on programme alors une boucle avec un test, qui recopie dans un deuxième fichier tous les éléments du premier sauf un ; et il faut ensuite recopier intégralement le deuxième dans le premier… Ouf.
- l'autre consiste à passer par un ou plusieurs tableaux. On recopie l'intégralité du fichier de départ dans un tableau (en mémoire vive) et on ne manipule ensuite que ce tableau, qu'on recopie à la fin dans le fichier d'origine.

Les avantages de la seconde technique sont nombreux :

- la **rapidité** : les accès en mémoire vive sont des milliers de fois plus rapides (nanosecondes) que les accès aux mémoires de masse (millisecondes au mieux pour un disque dur)
- la **facilité de programmation** : ; bien qu'il faille écrire en plus les instructions de recopie du fichier dans le tableau, pour peu qu'on doive tripoter ces informations dans tous les sens, c'est largement plus facile de faire cela avec un tableau qu'avec des fichiers.

Pourquoi, alors, demanderez-vous haletants, ne fait-on pas cela à tous les coups ? Y a-t-il des cas où il vaut mieux en rester aux fichiers et ne pas passer par des tableaux ?

- la recopie d'un gros fichier en mémoire vive consomme des ressources qui peuvent atteindre des dimensions considérables. Et au final, le gain de rapidité d'exécution risque de ne pas être évident.
- si le fichier contient des données de type non homogènes (chaînes, numériques, etc.) cela risque d'être coton pour le stocker dans un tableau : il va falloir déclarer plusieurs tableaux, dont le maniement au final peut être aussi lourd que celui des fichiers de départ. Ou alors, il faut utiliser une ruse qu'offrent certains langages (mais pas tous) : créer des types de variables personnalisés, composés d'un " collage " de plusieurs types existants (10 caractères, puis un numérique, puis 15 caractères, etc.) Cette technique est un peu acrobatique. Bien qu'elle ne soit pas vraiment difficile, elle exige tout de même une certaine aisance.

Je rappelle qu'a priori, les fichiers textes ne sont pas là pour servir de bases de données (en tout cas dans des langages de programmation récents). Donc, on aura peu de chances (ou de risques) de se trouver devant des immenses fichiers textes, ne pouvant pas être mis en tableau. Mais ce n'est pas une règle : il faut donc bien comprendre les tenants et les aboutissants de chaque stratégie. Et pour cela, quoi de mieux que la pratique ?

IX Procédures et fonctions

1. Sous-procédures

Une application, surtout si elle est longue, a toutes les chances de devoir procéder aux mêmes traitements, ou à des traitements similaires, à plusieurs endroits de son déroulement. Par exemple, la saisie (et le contrôle qu'elle implique) d'une réponse par oui ou par non, peuvent être répétés dix fois à des moments différents de la même application.

La manière la plus immédiate – et la moins habile – de résoudre la question est bien entendu de répéter le code correspondant autant de fois que nécessaire. Ainsi, la structure peut paraître simple. Mais elle est inutilement lourdingue, et en réalité, pour peu que le programme soit joufflu, il peut devenir parfaitement illisible.

Il faut donc opter pour une autre stratégie, qui consiste à séparer ce traitement du corps du programme et à appeler ces instructions (qui ne figurent donc plus qu'en un seul exemplaire) à chaque fois qu'on en a besoin.

Le corps du programme s'appelle alors la **procédure principale**, et ces groupes d'instructions auxquels on a recours s'appellent des **sous-procédures**.

Reprenons cet exemple de question à laquelle l'utilisateur doit répondre par oui ou par non.

Mauvaise Structure

```
...
Ecrire "Etes-vous marié ?"
Rep = ""
TantQue Rep <> " Oui" et Rep <> "Non"
Ecrire "Tapez Oui ou Non"
Lire Rep
FinTantQue
Ecrire "Avez-vous des enfants ?"
Rep = ""
TantQue Rep <> "Oui" et Rep <> "Non"
Ecrire "Tapez Oui ou Non"
Lire Rep
FinTantQue
```

Bonne Structure

```
Ecrire "Etes-vous marié ?"
RéponseOuiNon()
...
Ecrire "Avez-vous des enfants ?"
RéponseOuiNon()
...
Procédure RéponseOuiNon()
Rep = ""
TantQue Rep <> " Oui" et Rep <> "Non"
Ecrire "Tapez Oui ou Non"
Lire Rep
FinTantQue
Fin Procédure
```

Dans ce cas, on se contente donc d'appeler des lignes de codes qui se répètent à l'identique. Mais on peut avoir des cas beaucoup plus rigolos.

2. Passage de paramètres

Reprenons l'exemple qui précède et analysons-le. Nous écrivons un message, puis appelons la procédure pour poser une question ; puis plus tard, on écrit un autre message, et on relance la procédure pour poser la même question, etc. C'est une démarche acceptable, mais qui peut encore être améliorée : puisqu'avant chaque question, on doit écrire un message, autant que cette écriture du message figure directement dans la procédure appelée. Cela implique deux choses :

- lorsqu'on appelle la procédure, on doit lui préciser quel message elle doit afficher avant de lire la réponse
- la procédure doit être " prévenue " qu'elle recevra un message, et être capable de le récupérer pour l'afficher.

En langage algorithmique, on dira que le message est un **paramètre**. Comme le paramètre (le message) est transmis de la procédure principale vers la sous-procédure, il s'agit d'un paramètre **en entrée**. Voilà comment l'affaire se présente :

La procédure sera déclarée comme suit :

```
Procédure RéponseOuiNon(Msg en Caractère)
...
```

Il y a donc dorénavant une variable, Msg, dont on précise le type, et qui signale à le procédure qu'un paramètre peut lui être envoyé. A présent, le corps de la procédure sera :

```
Procédure RéponseOuiNon(Msg en Caractère)
Ecrire Msg
Rep = ""
TantQue Rep <> " Oui" et Rep <> "Non"
Ecrire "Tapez Oui ou Non"
Lire Rep
FinTantQue
Fin Procédure
```

La procédure principale devient alors :

```
RéponseOuiNon("Etes-vous marié ?")
...
RéponseOuiNon("Avez-vous des enfants ?")
...
```

Et voilà le travail. Là, on a passé qu'un seul paramètre en entrée, mais on peut en passer autant qu'on veut ; mais ne soyons pas gourmands, il suffit d'en passer autant qu'on en a besoin.

D'un autre côté, on ne s'est guère préoccupé jusque là de récupérer le résultat de notre sous-procédure, à savoir la réponse à la question posée. Examinons maintenant ce problème.

Une première solution serait la suivante : juste après l'appel de la procédure, on teste la variable Rep :

Si Rep = "Oui" **alors** …

L'inconvénient de cette technique, c'est que la variable Rep doit pouvoir conserver son contenu en voyageant d'une procédure à l'autre. On en reparlera plus loin mais ce genre de variable est extrêmement gourmand en mémoire vive. Donc, si on appelle comme cela plusieurs procédures, avec plein de variables qu'on veut récupérer, cela risque de coincer sérieusement à l'exécution.

La solution la plus économique en ressources consiste à transmettre la valeur de la variable Rep de la sous-procédure vers la procédure principale, exactement comme on transmettait jusque là la valeur de la variable Msg de la procédure principale vers la sous-procédure. On introduit donc un second paramètre, mais qui est maintenant un **paramètre en sortie**.

La déclaration de la procédure devient :

```
Procédure RéponseOuiNon(Msg en Caractère, Rep en Booléen)
L'appel devient quant à lui :
RéponseOuiNon("Etes-vous marié ?", toto)
si toto = "Oui" alors
...
RéponseOuiNon("Avez-vous des enfants ?", tata)
si tata = "Oui" alors
...
```

Là encore, rien n'empêche une sous-procédure de renvoyer plusieurs paramètres en sortie, vous avez toute liberté !

3. Fonctions personnalisées

Si vous n'avez qu'un seul paramètre à renvoyer en sortie, alors vous pourrez opter pour une solution encore plus légère : créer, plutôt qu'une sous procédure, une fonction personnalisée.

Ce type de fonction, comme une fonction normale, renvoie une valeur. Mais cette fois, c'est vous qui créez la fonction, en indiquant le nombre et le type des paramètres en entrée, le mode de traitement et la valeur qui doit être renvoyée en sortie.

Imaginons qu'à plusieurs reprises au cours d'une application, on ait besoin de calculer la moyenne d'un tableau de notes, le nombre de notes étant susceptible de varier d'une fois sur l'autre. Plutôt qu'écrire plusieurs boucles de calcul, qui à chaque fois risquent de se ressembler, il sera préférable de créer une fonction pour calculer cette moyenne. Cette fonction admettra deux arguments, qui lui sont indispensables pour effectuer ses calculs :

Le tableau lui-même
Le nombre d'éléments du tableau

Cela donnera, en pseudo-code, la chose suivante :

```
Fonction Moy(Tab en tableau numérique, n en entier)
Som = 0
Pour i = 1 à n
Som = Som + Tab(i)
i suivant
m = som / n
Renvoyer m
Fin Fonction
```

On remarque au passage l'apparition d'un nouveau mot-clé : Renvoyer, qui indique quelle valeur doit prendre la fonction.

Quant au programme principal, quelques lignes pourraient en être :

```
Tableau Notes en Numérique
....
moyenne = Moy(notes, nbnotes)
écrire "la moyenne est : ", moyenne
...
```

D'une manière générale, la création de fonctions est un puissant outil de programmation, qui ne possède que des avantages : on économise des lignes de programmation, on rend le déboguage plus facile, et on ne gaspille aucune ressource de la machine. Pensez-y donc sérieusement à chaque fois que vous devez créer une application un peu joufflue (un projet, par exemple).

4. Une logique vicelarde : la programmation récursive

Vous savez comment sont les informaticiens : on ne peut pas leur donner quoi que ce soit sans qu'ils essayent de jouer avec, et le pire, c'est qu'ils y réussissent.

Cette technique des fonctions personnalisées a donné lieu à la floraison d'une logique un peu particulière, destinée en particulier à traiter certains problèmes mathématiques (ou de jeux) : la programmation récursive. Pour vous expliquer de quoi il retourne, nous allons reprendre un exemple cher à vos cœurs : le calcul d'une factorielle (là, je sentais que j'allais encore me faire des copains).

Rappelez-vous : la formule de calcul de la factorielle d'un nombre n s'écrit :

N ! = 1 x 2 x 3 x … x n

Nous avions programmé cela aussi sec avec une boucle Pour, et roule Raoul. Mais une autre manière de voir les choses, ni plus juste, ni moins juste, serait de dire que :

N ! = n x (n-1) !

En bon français : la factorielle d'un nombre, c'est ce nombre multiplié par la factorielle du nombre précédent. Encore une fois, c'est une manière ni plus juste ni moins juste de présenter les choses ; c'est simplement une manière différente.

Si l'on doit programmer cela, on peut alors imaginer une fonction Fact, chargée de calculer la factorielle. Cette fonction effectue la multiplication du nombre passé en argument par la factorielle du nombre précédent. Et cette factorielle du nombre précédent va bien entendu être elle-même calculée par la fonction Fact.

Autrement dit, on va créer une fonction qui pour fournir son résultat, **va s'appeler elle-même un certain nombre de fois**. C'est cela, la récursivité.

Toutefois, il nous manque une chose pour finir : quand ces auto-appels de la fonction Fact vont-ils s'arrêter ? Cela n'aura-t-il donc jamais de fin ? Si, bien sûr, rassure-toi, ô public, la récursivité, ce n'est pas Les Feux de L'Amour. On s'arrête quand on arrive au nombre 1, pour lequel la factorielle est par définition 1.

Cela produit l'écriture suivante, un peu déconcertante certes, mais parfois très pratique :

```
Fonction Fact (N en Numérique)
Si N = 1 alors
Renvoyer 1
Sinon
Renvoyer Fact(N-1)
Finsi
Fin Fonction
```

Vous remarquerez que le processus récursif remplace en quelque sorte la boucle, c'est-à-dire un processus itératif. Et en plus, avec tous ces nouveaux mots qui riment, vous allez pouvoir écrire de très chouettes poèmes. Vous remarquerez aussi qu'on traite le problème à l'envers : on part du nombre, et on remonte à rebours jusqu'à 1 pour pouvoir calculer la factorielle. Cet effet de rebours est caractéristique de la programmation récursive.

Pour conclure sur la récursivité, trois remarques fondamentales.

- la programmation récursive, pour traiter certains problèmes, est **très économique pour le programmeur** ; elle permet de faire les choses correctement, en très peu de lignes de programmation.
- en revanche, elle est **très dispendieuse de ressources machine**. Car à l'exécution, la machine va être obligée de créer autant de variable temporaires que de " tours " de fonction en attente.
- Last but not least, et c'est le gag final, **tout problème formulé en termes récursifs peut également être formulé en termes itératifs !** Donc, si la programmation récursive peut faciliter la vie du programmeur, elle n'est pas indispensable. Mais ça me faisait tant plaisir de vous en parler que je n'ai pas pu résister...

5. Variables publiques et privées

L'existence de sous-procédures, de paramètres, pose le problème de la " durée de vie " des variables, ce qu'on appelle leur **portée**. Pour faire simple, une variable peut être déclarée :

- Comme privée, ou locale (c'est en général l'option par défaut). Cela signifie qu'une telle variable disparaît (et sa valeur avec) dès que prend fin la procédure ou elle a été créée.
- Comme publique, ou globale. Ce qui signifie qu'une telle variable est conservée intacte pour toute l'application, au-delà des ouvertures et fermetures de procédures.

La manière dont ces déclarations doivent être faites est évidemment fonction de chaque langage de programmation. En pseudo-code algorithmique, vous pourrez utiliser le mot-clé **Public** pour déclarer une variable publique :

Public Toto **en Entier**

Comment choisir entre déclarer une variable en Public ou en Privé ? C'est très simple : **les variables globales consomment énormément de ressources en mémoire**. En conséquence, le principe qui doit présider au choix entre variables publiques et privées doit être celui de **l'économie de moyens** : on ne déclare comme publiques que les variables qui doivent absolument l'être. Et chaque fois que possible, lorsqu'on crée une sous-procédure, on utilise le passage de paramètres par valeur plutôt que des variables publiques.

6. Algorithmes fonctionnels

Pour clore ce chapitre, quelques mots à propos de la structure générale d'une application. Celle-ci va couramment être formée d'une procédure principale, et de sous-procédures (qui vont au besoin elles-mêmes en appeler d'autres, etc.). L'exemple typique est celui d'un menu, ou d'un sommaire, qui " branche " sur différents traitements, donc différentes sous-procédures.

L'algorithme fonctionnel de l'application est la **représentation graphique** de cette structure générale, ayant comme objectif de faire comprendre d'un seul coup d'œil quelle procédure fait quoi, et quelle procédure appelle quelle autre. L'algorithme fonctionnel est donc en quelque sorte la représentation du squelette de l'application. Il se situe à un niveau plus général, plus abstrait, que l'algorithme normal, qui lui, détaille pas à pas les traitements effectués au sein de chaque procédure.

Dans la construction – et la compréhension – d'une application, les deux documents sont indispensables, et constituent deux étapes successives de l'élaboration d'un projet.

X Notions complémentaires

Une fois n'est pas coutume, ce chapitre ne sera l'objet d'aucun exercice. Cela ne veut pas dire pour autant que ce qui s'y trouve n'est pas intéressant.

Non mais des fois.

1- Programmation structurée

Petit retour sur une notion très rapidement survolée plus haut : celle de " programmation structurée ". En fait, nous avons jusqu'à présent, tels Monsieur Jourdain, fait de la programmation structurée sans le savoir. Aussi, plutôt qu'expliquer longuement en quoi cela consiste, je préfère passer directement à raconter en quoi cela ne consiste pas.

Dans certains langages (historiquement, ce sont généralement des langages anciens), les lignes de programmation portent des numéros. Les lignes sont ensuite exécutées dans l'ordre de leurs numéros. Jusqu'ici, en soi, pas de problème. Mais l'astuce est que tous ces langages, il existe une instruction de branchement, notée aller à en pseudo-code, instruction qui envoie directement le programme à la ligne spécifiée. Inversement, ce type de langage ne comporte pas d'instructions comme FinTantQue, ou FinSi, qui " ferment " un bloc.

Prenons l'exemple d'une structure " Si … Alors … Sinon "

Programmation Structurée

Si condition **Alors**
instructions 1
Sinon
instructions 2
FinSi

Programmation non structurée

1000 **Si** condition **Alors Aller En** 1100 **Sinon Aller En** 1200
1100 instruction 1
1110 etc.
1120 etc.
1190 **Aller en** 1400
1200 instruction 2
1210 etc.
1220 etc.
1350 **Aller en** 1400
1400 suite de l'algorithme

Vous voyez le topo : un programme écrit dans ce type de langages se présente comme une suite de branchements emmêlés les uns des les autres. D'une part, on ne peut pas dire que cela favorise la lisibilité du programme. D'autre part, c'est une source importante d'erreurs, car tôt ou tard on oublie un " aller à ", ou on un met un de trop, etc. A fortiori lorsqu'on complique un algorithme existant, cela peut devenir un jungle inextricable.

A l'inverse, la programmation structurée, surtout si l'on prend soin de rationaliser la présentation en mettant des lignes de commentaires et en pratiquant l'indentation, évite des erreurs, et se révèle sa structure logique de manière très claire.

Le danger est que si la plupart des langages de programmation utilisés sont structurés, ils offrent tout de même la plupart du temps la possibilité de pratiquer la programmation non structurée. Dans ce cas, les lignes ne sont pas désignées par des numéros, mais certaines peuvent être repérées par des noms (dits " étiquettes ") et on dispose d'une instruction de branchement.

Une règle d'hygiène absolue est de programmer systématiquement de manière structurée, sauf impératif contraire fixé par le langage.

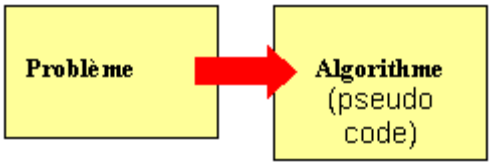
Autrement dit, même quand un langage vous offre une possibilité de faire des entorses à la programmation structurée, il ne faut s'en saisir sous aucun prétexte.

2. Interprétation et compilation

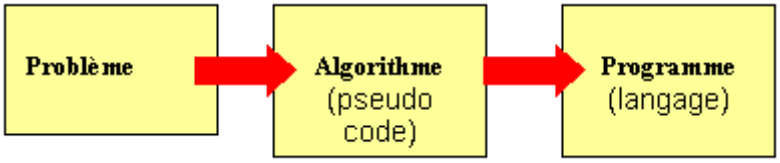
Avec ce paragraphe, on sort un peu de l'algorithmique proprement dite pour entrer dans le domaine plus technique de la réalisation pratique. Ou, si l'on préfère, ces dernières lignes sont l'apothéose, le bouquet final, l'extase ultime, la consécration grandiose, de ce cours.

En toute modestie, bien sûr.

Jusqu'ici, nous avons travaillé sur la première étape de la réalisation d'un programme :



Si l'algorithme est bien écrit, sans faute logique, l'étape suivante ne doit normalement poser aucun problème conceptuel. C'est une simple traduction que vous devez effectuer :



A partir de là, votre travail est virtuellement terminé (si l'on excepte l'inévitable phase de tests, corrections, etc.). Mais pour l'ordinateur, c'est là que les ennuis commencent. En effet, aucun ordinateur n'est en soi apte à exécuter les instructions telles qu'elles sont rédigées dans tel ou tel langage ; l'ordinateur, lui, ne comprend qu'un seul langage, qui est un langage codé en binaire (à la rigueur en hexadécimal) et qui s'appelle le langage machine (ou assembleur).

C'est à cela que sert un langage : à vous épargner la programmation en binaire (une pure horreur, vous vous en doutez) et vous permettre de vous faire comprendre de l'ordinateur d'une manière (relativement) lisible.

C'est pourquoi tout langage, à partir d'un programme écrit, doit obligatoirement procéder à une traduction en langage machine pour que ce programme soit exécutable.

Il existe deux stratégies de traduction, ces deux stratégies étant parfois disponibles au sein du même langage.

- le langage traduit les instructions au fur et à mesure qu'elles se présentent. Cela s'appelle la **compilation à la volée**, ou **l'interprétation**.
- le langage commence par traduire l'ensemble du programme en langage machine, constituant ainsi un deuxième programme (un deuxième fichier) distinct physiquement et logiquement du premier. Ensuite, et ensuite seulement, il exécute ce second programme. Cela s'appelle la **compilation**.

Il va de soi qu'un langage interprété est plus maniable : on peut exécuter directement son code au fur et à mesure qu'on le tape, sans passer à chaque fois par l'étape supplémentaire de la compilation. Mais il va aussi de soi qu'un programme compilé s'exécute beaucoup plus rapidement qu'un programme interprété : le gain est couramment d'un facteur 10, voire 20 ou plus.

Toute application destinée à un usage professionnel (ou même, tout simplement sérieux) est forcément une application compilée.